In this last lecture, we look at

- Numeric types and numerical calculations,
- Advanced aspects of C operators, and
- The standard C environment.
Choosing the proper data type.
Representational properties.
Computational issues.
Casts and mixed-type operations.
Sticky points and common errors.
Representation error.
**Integer or real?** For most problems obvious.

**Float vs. double.** Because a programmer can combine types float and double freely in expressions, most of the time, it does not matter which real type is used. Sometimes the degree of precision required for the data dictates the use of double. Since all the functions in the math library expect double arguments and return double results, most programmers just find it easier to declare all real variables as double.

However, if you are processing large amounts of data and precision is not important, then float variables use only half as much space as doubles and an int even less in some cases. Integer calculations are faster than real calculations and float values are faster than double values.
For integers: signed, unsigned, short and long.
- Max short signed: 32767
- Max long signed: 2 147 483 647
- Max long unsigned: 4 294 967 295

For float the minimum value range is

\[1.175 \times 10^{-38} \ldots 3.402 \times 10^{38}.\]

For double:

\[2.225 \times 10^{-308} \ldots 1.797 \times 10^{308}.\]
Numeric: Computational issues 1

- Integer division is not the same as division using real numbers; any remainder from an integer division is forgotten. The remainder must be computed by using a modulus operator(\%).

- Sometimes the underlying computer hardware does not support floating-point arithmetic, in which case floating point representation and computation must be emulated by software. This is slow.
C supports mixed-type arithmetic. When two values of differing types are used with an operator, the value with less precision automatically is coerced to the more precise representation.

If an integer is combined with float or a double in an expression, the integer operand always is converted to the type of the floating-point operand before the operation is performed. The result is a floating-point value.

A type conversion may be safe, in that it will cause no loss of information, or it may be unsafe. Knowing when a type conversion can be used safely is important. However, sometimes an unsafe conversion is exactly what the programmer needs.

An explicit type cast must be used to perform real division with integer operands.
Using the wrong conversion specifier in a format can cause input or output to appear as garbage. Default length, short, and long integers have different conversion codes, as do signed and unsigned integers.

When using reals, there is no way to tell from the printed output whether a value came from a double or a float variable. If you specify a format such as %.10f, you might see 10 nonzero digits printed, but that does not mean that all are accurate. If the number came from float variable, the eighth through tenth digits usually will be garbage.

Some systems use 2 bytes to represent int, others use 4 bytes. This makes the portability of code a nightmare. Errors due to integer sizes are among the hardest to find because of the ever-present automatic size conversions all C translators perform.
Types float and double are *approximate representations* for real numbers, but with differing precision. Consider the code below:

```c
float w = 4.4;   double x = 4.4;

printf(" Is x == (double) w? %i \n", (x == (double)w) );
printf(" Is (float)x == w? %i \n", ((float)x == w) );
```

The output may be unexpected, if you forgot that the two numbers are represented with limited, and different, precision and that the `==` operator tests for exact bit-by-bit equality:

- Is x == (double)w? 0
- Is (float)x == w? 1
Numeric: Representation error II

When a more-precise value is cast to the less-precise type, the extra bits are truncated and the numbers are exactly equal. When a shorter value is cast to the longer type, it is lengthened by adding zero bits at the end of the mantissa, not by recomputing the additional bit values. In general, these zeros are not equal to the meaningful bits in the double value.

Computation also can introduce representational error, as shown by the next code fragment:

```c
float w;
    double x, y = 11.0, z = 9.0;
    x = z * (y / z);
    w = y-x;
```
The result is as expected, \( w = 0, \ x = 11.000 \). But change the starting values \( y = 15.0 \) and \( z = 11.0 \), and the result is

\[
\begin{align*}
w &= 1.77635e - 15, \\
x &= 15.000.
\end{align*}
\]

Why does this happen? The answer to a floating-point division has a fractional part that is represented with as much precision as the hardware will allow. However, the precision is not infinite and there is a tiny amount of truncation error after most calculations. Therefore, the answer to \( y/z \) may have error in it, and that error is increased when we multiply by \( z \).
When are two floating-point numbers equal? The answer is that they should be called equal if both are approximations for the same real number, even if one approximation has more precision than the other.

Practical problems often require comparing a calculated value to a specific constant or setpoint or comparing two calculated values that should be equal. Such a comparison is not as simple as it seems, because even simple computations with small floating-point values can have results that differ from the mathematically correct versions.

We can get around this comparison problem by comparing the difference of the two numbers to a preset epsilon value. For any given application, we can choose a value of epsilon that is slightly smaller than the smallest measurable difference in the data.

This kind of comparison can be made with a single if command, if we use \( \text{fabs()} \) function from the math library:
double epsilon = 1.0e-3;
double number, target;

if (fabs(number - target) < epsilon)
    /* then we consider that number == target */
else
    /* we consider the values different; */
• **Integer overflow and wrap.** Suppose that the 2-byte integer variable $k$ contains the number 32300 and you enter a loop that adds 100 to $k$ seven times. The value stored in $k$ would be, in turn, 32400, ..., 32700, −32736, ..., −32536. The value has wrapped around and become negative, but that does not stop the computer. This can be a potential security hole allowing buffer overflow attacks.

• **Floating-point overflow and infinity.** The phenomenon of wrap is unique to integers; floating-point overflow is handled differently. The IEEE floating-point standard defines a special bit pattern, called infinity, that will result if overflow occurs during a computation. The constant HUGE_VAL, defined in math.h, is set to be the infinity value on each local system. Therefore, one way an overflow can be detected is by comparing a result to HUGE_VAL or -HUGE_VAL.
Underflow. Underflow occurs, when the magnitude of the number falls below the smallest number in the representable range. This can, of course, happen only for real numbers. For real numbers, underflow happens when a value is generated that has a 0 exponent and a nonzero mantissa. Such a number is referred to as *denormalized*, which means that all significant bits have been shifted to the right and the number is less than the lowest number specified by the standard. Some systems will generate the 0 value when the lower bound has been reached. Others still use the denormalized values. Underflow can result from several kinds of computations:

- Dividing a number by a very large number or repeated division.
- Multiplying a small number by a near-zero number, which has the same effect as dividing by a very large number.
- Subtracting two values that are near the smallest representable float and ought to be equal but are not quite equal because of round-off error.
If you add a small float number to a large one, and their exponents differ by more than $10^7$, the addition likely will have no effect. The answer will be the same large number that you started with.

A special value called **NAN**, which stands for "not a number", can be generated through operations such as $0/0$. This is another special bit pattern that does not correspond to a real value. The IEEE standard defines that any further operation attempted using a NaN or Infinity as an operand will return the same same value.
We deal with the following:

- Assignment combinations.
- Lazy evaluation.
- Evaluation order and side-effect operators.
- Conditional operator.
All the assignment-combination operators have the same very low precedence and associate right to left. Consider for example the expression
\[ t /= n -= m *= k += 7; \]

The evaluation starts with \( k = k + 7 \) and continues from right to left. So + is made before * even if in normal expressions the precedence of multiplication is higher than the precedence of addition.
With lazy evaluation, when we skip, we skip the right operand.
This is not confusing, when the right operand is only a simple variable. However, sometimes it is an expression with several operators.
Consider an example:
\[ y = a < 10 \; \text{||} \; a \geq 2 \times b \; \&\& \; b \neq 1; \]
The left operand is \( a < 10 \) and the right operand is \( a \geq 2 \times b \; b! = 1 \).
If \( a = 7 \), we skip all after || and \( y \) get the value 1. If \( a = 17 \) and \( b = 20 \) the expression after && is not evaluated and \( y \) becomes 0.
A usual assumption is that && is executed first, because it has higher precedence. Although precedence controls the construction of the parse tree, precedence simply is not considered when the tree is evaluated.
Sometimes lazy evaluation can substantially improve the efficiency of a program. A much more important use of for skipping is to avoid evaluating parts of an expression that would cause machine crashes or other kinds of troubles.

Assume that we want to divide a number \( x \), compare the answer to a minimum value, and do an error procedure if the answer is less than the minimum. But it is possible for \( x \) to be 0 and that must be checked. We can avoid a division-by-0 error and do the computation and comparison in one expression by using a *guard* before the division:

\[
\text{if } (x \neq 0) \land \land \text{ total } / x < \text{ minimum} \text{ do_error();}
\]

The skipping may happen in the middle of an expression. Consider

\[
y = a < 0 \lor a > b \land b > c \lor b > 10;
\]

for \( a = 3 \) and \( b = 17 \). The subexpression \( b > c \) is not evaluated, all the other are and \( y \) becomes 1.
When used in isolation, the increment and decrement operators are convenient and relatively free of complication. When side-effect operators are used in long, complex expressions, they create the kind of complexity that fosters errors. If such an operator is used in the middle of a logical expression, it may be executed sometimes but skipped at other times.

A second problem with side-effect operators relates to the order in which the parts of an expression are evaluated. Recall that the evaluation order has nothing to do with precedence order. We have stated that logical operators are executed left to right. This is also true of two other kinds of sequencing operators: the conditional operator ?...: and the comma ( for example, for (i = 1, j = 1; ...)). Most other operators can be evaluated right-side first or left-side first.
This leads to one important warning: If an expression contains a side-effect operator that changes the value of a variable $V$, *do not use* $V$ anywhere else in the expression. The side-effect could happen either before or after the value of $V$ is used elsewhere in the expression and the outcome is unpredictable.
Even if the evaluation in an expression with the conditional operator starts by calculating true or false value, the value of the entire conditional operator, in general, will not be true or false.

If the condition contains any postincrement operators, the increments must be done before evaluating the true clause or the false clause. Therefore, it is “safe” to use postincrement in the condition.
The following libraries are useful to know:

- limits.h
- float.h
- ctype.h
- math.h
- stdio.h
- stdlib.h
- string.h
- time.h
• Number of bits in a character: CHAR_BIT.
• Type character: CHAR_MAX, CHAR_MIN.
• Signed and unsigned characters: SCAR_MAX, SCAR_MIN, UCHAR_MAX.
• Signed and unsigned short integers: SHRT_MAX, SCHRT_MIN, USHTR_MAX.
• Signed and unsigned integers: INT_MAX, INT_MIN, UINT_MAX,
• Signed and unsigned long integers: LONG_MAX, LONG_MIN, ULONG_MAX,
- FLT_RADIX: the value of radix (or base).
- FLT_ROUNDS: rounding mode.
- FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON: minimum $x$ such that $1.0 + x \neq 1$.
- etc (decimal digit precision, number radix digits in the mantissa, minimum normalized positive number,...).
- int isalnum(int ch);
- int isalpha(int ch);
- int isspace(int ch);
- int tolower(int ch);
- etc.
- trigonometric functions,
- logarithms and powers,
- manipulating number representations (ceil, floor, fabs, fmod, etc).
Already known and used. Some interesting things:

- FOPEN_MAX: the number of streams that can be open simultaneously.
- FILENAME_MAX.
- rename(const oldname, const newname).
- remove(char filename).
- int setvbuf(FILE str, char buf, int bufmode, size_t size): Changes the buffer to be used for I/O operations with the specified stream. The function allows to specify the mode and size for the buffer. This function should be called once the file associated with the stream has already been opened but before any input or output operation has taken place. The size of the buffer is specified by the size parameter in bytes.
- int abs(int x).
- srand, rand.
- void * bsearch( const void* key, const void* base, size_t count, size_t size, int (*compar)(const void* key, const void* value)).

  void * qsort( const void* base, size_t count, size_t size, int (*compar)(const void* e1, const void* e2)).

- malloc, calloc, free, realloc.
- String to number conversion functions.
char getenv(const char* name): Retrieves a C string containing the value of the environment variable whose name is specified as argument. If the requested variable is not part of the environment list, the function returns a NULL pointer. The string pointed by the pointer returned by this function shall not be modified by the program.
- `time_t`, `clock_t`.
- `struct tm`: a structure representation of the time.